Improving Mobile Augmented Reality User Experience on Smartphones

---

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
in
Computer Science and Software Engineering
by
Charles ZhouXiao Han

University of Canterbury
2010

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my appreciations to Dr. R. Mukundan and Prof. Mark Billinghurst. Both supervisors have guided me greatly in my research project. I could not finish my thesis without their advise and support.

I am grateful to the staff at the HIT Lab NZ, especially Robert Ramsay who spent lots of time to help me troubleshoot hardware and software problems.

Last but not least, I want to thank you my parents and my wife Li Zhou for their unconditional love and support.

Abstract

This thesis focuses on improving the user experience for computer vision-based Augmented Reality (AR) applications on smartphones. The first part shows our proposed methods to enhance image binarisation. This improves the marker detection results in mobile AR applications. The comparisons of the original ARToolKit binarization method, our proposed histogram-based automatic thresholding and our histogram equalization based thresholding show that the histogram-based automatic thresholding produces a relatively better result under extreme and normal lighting conditions but slightly reduces the ARToolKit framerate. The second part introduces a new fast painterly rendering algorithm which produces an immersive experience for mobile AR users. The proposed algorithm has low complexity and achieves a real-time performance on smartphones. In addition, this study has carried out a preliminary experiment comparing mobile GPU-based image processing algorithms with CPU-based equivalent on smartphones. The result indicates that the GPU-based implementations perform better than the CPU when processing relatively large sized images.

Chapter I

Introduction

1.1 Overview of Augmented Reality

Azuma[2] defines Augmented Reality (AR) as a state between reality and virtual reality where virtual objects are overlaid on the real world. One purpose of AR applications is to enhance their real world with artificial information which can help end users to understand the surrounding environment better. AR applications can also change how users interact with computer devices [4].

One of the key components of an AR system is tracking technology that can be used to specify the users viewpoint. Current AR tracking technologies include computer vision-based tracking, inertial tracking and magnetic tracking. Among these technologies, computer vision-based tracking may promise the most pixel-perfect alignment of augmented reality as well as relatively simple hardware setup. It only requires a digital camera, a computer and software that uses image processing and computer vision methods to analyse captured video streams and calculates reference object position and orientation.

Early AR applications were developed on desktop and laptop computers. Users wore Head Mounted Displays (HMDs) attached to the computers to view augmented images [17]. Figure 1.1 shows an example of such setup. However, HMDs are still expensive for every day users. Current developments in hardware and software technology have enabled AR applications to run on commodity portable devices such as camera-equipped smartphones.

A smartphone [1] is defined as a mobile phone that has advanced computing power close to a mobile PC. Modern smartphones have become multi-purpose

[1] http://en.wikipedia.org/wiki/Smartphone

Figure 1.1: Head Mounted Display used in the ARToolKit. (*Photograph courtesy of the Human Interface Technology Laboratory (HIT Lab) at the University of Washington*)

devices rather than just answering phone calls and text messaging. End users have started using smartphones every day for web browsing, email and gaming. The line between a phone and a computer is becoming blurred. Advanced smartphones such as the Apple's iPhones [2] and the NOKIA's N900 are now equipped with dedicated graphics processors to enable them to run graphics intensive software applications that in the past would only have run on desktop computers.

1.2 Examples of AR Applications

Researchers and developers have been experimenting and building AR applications in many different domains, for instance education, gaming and advertising:

Education

The Mixed Reality Book (figure 1.2) project uses AR technology to ex-

[2] http://en.wikipedia.org/wiki/IPhone

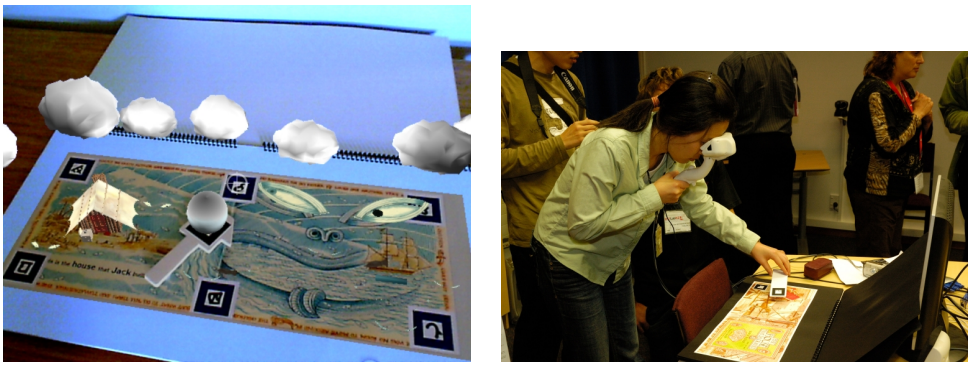


Figure 1.2: Mixed Reality Book project developed by the HIT Lab NZ. (*Photograph courtesy of the HIT Lab at the University of Canterbury, New Zealand*)[11]

tend a physical book by adding virtual graphics, animation and audio, which provides readers extra information beyond text.

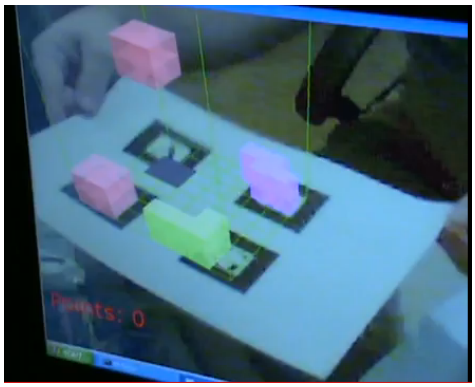
Gaming

ARTetris (figure 1.3) was developed by postgraduate students from the University of Canterbury to study and incorporate physical metaphors in AR interaction. The game uses AR technology to create a three dimensional volume and Tetris blocks. The player can rotate a paper marker to view different angles of the 3D volume. The player also uses a physical cube with markers attached to control the blocks, e.g. moving a block or rotating a block.

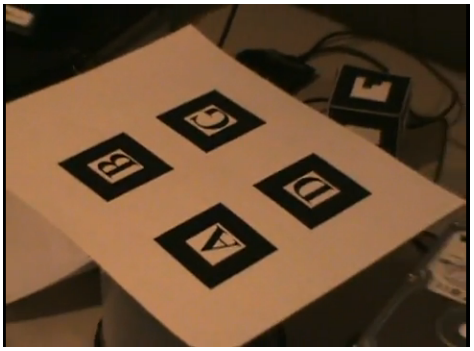
The AR Tennis (figure 1.4) is an AR game developed for smartphones. The game was for two players and each player used a smartphone to serve and return a virtual tennis ball.

Advertising

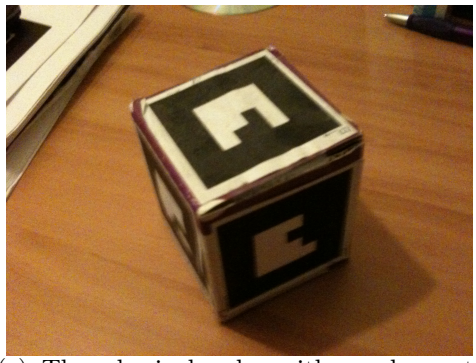
A mobile AR Advertising (figure 1.5) application was developed for the Wellington zoo to display 3D animal models on smartphones. Software was downloaded to a smartphone and displayed a 3D animal model when the phone's camera was pointed at a special tracking symbol printed in newspaper.



(a) Computer screen



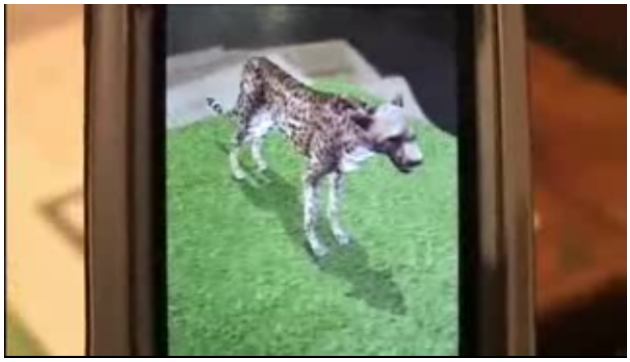
(b) The base marker



(c) The physical cube with markers attached

Figure 1.3: The 3D AR Tetris Project. (*Photograph by the author*)

Figure 1.4: AR Tennis. (*Photograph courtesy of the HIT Lab at the University of Canterbury, New Zealand*)[13]



(a) 3D animal model

(b) A special marker printed on newspapers

Figure 1.5: Mobile AR Advertising. (*Photograph courtesy of the HIT Lab at the University of Canterbury, New Zealand*)

Figure 1.6: A sample ARToolKit marker

1.3 Computer Vision-Based Fiducial Marker Tracking

Fiducial marker tracking uses a specially designed marker as a reference object in video stream and employs computer vision techniques to compute camera's coordinates in relation to the marker. ARToolKit [17] is a marker based tracking library originally developed by Dr. Hirokazu Kato. Its source code has been freely available for research and education purposes.

1.3.1 Introducing ARToolKit

ARToolKit uses a black and white marker as shown in Figure 1.6. Its square shape and black colour are designed to be easily recognised by computer vision algorithms. The symbol in the centre of the marker is its identity and is used to distinguish from other square objects in the video stream.

ARToolKit requires camera calibration to generate the camera's intrinsics parameters as well as distortion factors which form a projection matrix (1.1). The matrix can be used to translate a world coordinate to the image coordinate. The white and black marker is converted into a 16 by 16 grey scale image and is saved as a data file for later use.

$$A = \begin{bmatrix} sf_x & 0 & u \\ 0 & sf_y & v \\ 0 & 0 & 1 \end{bmatrix} \tag{1.1}$$

Figure 1.7 shows main steps of ARToolKit:

1. ARToolKit thresholds an input frame into a binary image and uses a labelling algorithm to find any possible square shape objects. Objects

that are too large or too small will be rejected immediately based on constant values.

2. For each labelled object, edges and corner points are detected. Image coordinates of the four corner points are used to form line equations (1.2) which represents the two perpendicular sides of the square marker.

$$a_1x_1 + b_1y_1 + c_1 = 0, a_2x_2 + b_2y_2 + c_2 = 0 \tag{1.2}$$

3. The detected marker's interior pattern is extracted and is normalised for template matching with the previously saved pattern. The equation (1.1) is used to translate and scale the captured marker pattern to the image space.

4. By using the corner points in the marker's coordinates and the points in the image screen coordinates , an iterative process is taken to find the closest rotation and translation components in the camera's extrinsic parameter (1.3).

$$\begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} = \begin{bmatrix} R_{11} & R_{12} & R_{13} & T_1 \\ R_{21} & R_{22} & R_{23} & T_2 \\ R_{31} & R_{32} & R_{33} & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_M \\ Y_M \\ Z_M \\ 1 \end{bmatrix} \tag{1.3}$$

5. In the final step OpenGL uses the estimated transformation matrix (1.3) to overlay a virtual object on the video frames.

1.3.2 Mobile AR Limitations

The mobile version of ARToolKit has the same inherent limitations of its desktop version but also has its own unique issues:

1. A desktop based AR setup is usually located in an indoor environment and the lighting condition can be adjusted accordingly. However,

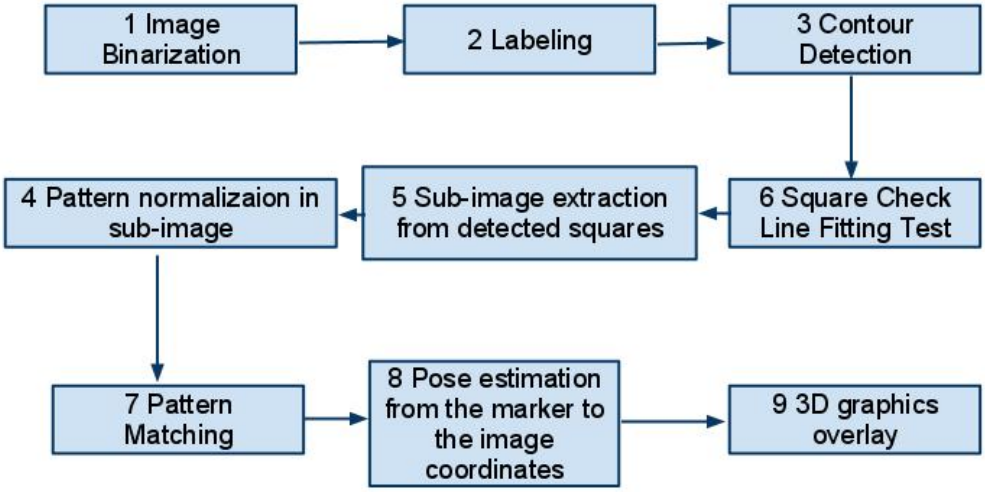


Figure 1.7: The ARToolKit workflow

mobile AR applications allow users to move around locations with different lighting conditions. It would be useful if the applications were able to adjust their image thresholding value accordingly. Currently the ARToolKit uses a fixed threholding value and can not work under extreme lighting conditions such as dimmed or bright environments.

2. Images taken by AR applications are usually not stabilized which results in poor marker detection and produces blurred rendering. Especially, mobile AR applications commonly use smartphones equipped with an integrated camera. It is difficult for a user to hold the smartphone firmly.

3. The ARToolKit markers must be always visible in video frames because the ARToolKit algorithm uses corner points to determine parallel lines and further tests the lines to form a rectangle area.

This project focuses on solving the first issue because computer vision based AR systems are easily affected by illuminations which can result a poor user experience.

1.4 Motivation of Improving Mobile AR

A current research trend is to bring AR applications onto smartphones to increase the level of accessibility of AR applications. Despite the computing power constraint and short battery life, smartphones have the following advantages for running AR applications:

Low Cost

Smartphones have become cheaper as the cost of electronic hardware reduces and so more consumers will be able to own smartphones.

Integrated User Interface

AR applications require optical cameras and a reasonably large display. Smartphones have integrated these two components together. Users will be able to easily hold smartphones to capture images and view augmented video streams at the same time.

Mobility of AR Applications

Smartphones are smaller than desktop or laptop computers. This allows users to easily carry smartphones and use its network connection to download AR content.

Smartphones are manufactured with embedded CPU chips which have less computing power than standard desktop PCs and laptops because embedded electronic components have physical limitations like over heating and small size of the circuit boards. To overcome these issues and enable AR applications on smartphones algorithms and methods need to be optimized.

Researchers have proposed a client/server architecture to offload computing power to the server. [1] uses a mobile phone to capture images and send them to a server which processes the incoming images, renders the AR content onto it and sends the final augmented image back to the phone. This type of method offloads the computational expense to backend computers but requires a private network setup which limits the accessibility of AR applications. In addition, there can be a significant latency due to network delays.

Recent research shows that modern graphics processing units (GPUs) can be programmed and could be used for enhancing general purpose computing. Researchers have also found that GPUs are suitable for computer image processing because many basic image processing algorithms could be executed parallel. GPUCV [5] is a computer vision library using desktop GPU. With the release of embedded graphics processors on smartphones it is possible to enhance AR applications by running some of the image computation on the graphics processor.

1.5 Research Objectives

This research focuses on enhancing computer vision and image processing methods on smartphones because vision-based AR applications rely heavily on computer vision techniques. An embedded version of ARToolKit has been built for ARM (Advanced RISC Machines) processors, which is the processor used in most commodity smartphones. This project uses NOKIA's N900 as a test platform. There are three research objectives in this thesis:

1. Explore and evaluate CPU and GPU-based computer vision algorithms on smartphones.

2. Enhance the image binarisation method to improve mobile AR marker detection process under different lighting environment.

3. Enhance rendering effects of mobile AR applications to provide users with a more immersive experience.

1.6 Outline of the Thesis

Chapter 2

Introduces mobile Graphics Processing Units and its importance on smartphones. Discusses GPU's limitations for image processing.

Chapter 3

Discusses the experimental results of comparing CPU and GPU-based image processing algorithms on smartphones.

Chapter 4

Demonstrates new methods for improving the marker detection process of ARToolKit using image histogram and histogram equalization. A new GPU-based histogram generation method is also introduced. Finally a experiment was run to compare the new methods integrated into ARToolKit with the original implementation.

Chapter 5

Presents our new real-time painterly rendering algorithm for mobile AR applications. Our GPU based edge enhancements are also presented.

Chapter II

Computer Vision and Image Processing on Graphics Processing Units (GPUs)

This chapter introduces programmable GPUs on smartphones and their importance. In addition, it describes previous related research in GPU-based general-purpose computing and GPU-based computer vision and image processing. Some known limitations of GPU-based image processing are discussed.

2.1 GPU Hardware and Software

Modern GPUs are primarily designed for accelerating computer graphics rendering. They have a parallel computing architecture which allows a single instruction to process a set of data simultaneously. GPUs were initially available on desktop computers and were widely used by computer games and specialized industrial design software for fast graphics processing. Since then the transistor's size has been reduced dramatically and GPUs are now available on laptop computers and even high-end consumer smartphones are also equipped with embedded GPUs. There is a demand for mobile GPUs because they improve the performance of portable computer games which are one of the most popular types of software running on mobile devices today. For example, Apple iPhone games are the most downloaded piece of mobile phone software [1].

Early versions of GPUs had a fixed rendering pipeline which takes in vertex information and renders pixels on a screen. Starting from 2001 GPU manufactures introduced programmable shaders which allow developers to

[1] http://techcrunch.com/2010/02/25/iphone-games-what-sells-distimo/

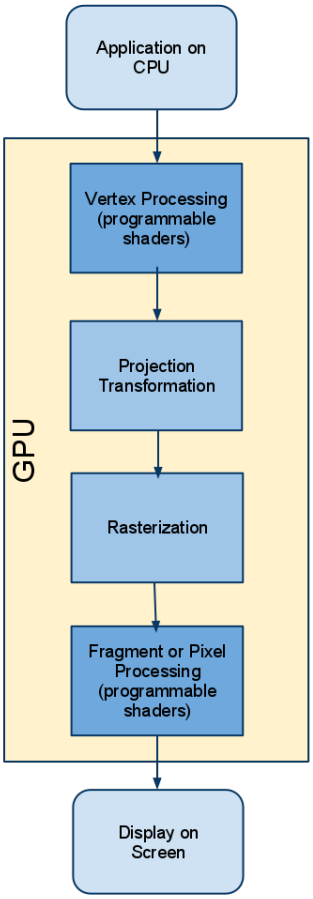


Figure 2.1: The simplified GPU rendering pipeline

write their own instructions to manipulate vertices and pixels. The programmable shaders can enable video games and cartoon movies to create special visual effects.

Figure 2.1 illustrates the simplified programmable GPU's rendering pipeline. For example, an application sends data for triangle vertices information to the GPU. The GPU processes the vertices based on user-defined shader instructions. The shader processes the colour and position of vertices. Then a sequence of transformations are applied to the vertices for correct display in 3D coordinates. Next, the fragment shaders process each pixel to calculate the colour before displaying it on the screen. Finally, all processed data will be sent to the device's screen.

There are two major types of shaders, *Vertex shaders* and *Fragment shaders* , also known as *Pixel Shaders*. Vertex shaders manipulate vertex attributes, such as position, colour and texture coordinates. Fragment shaders are used to modify pixel and texture colours before being drawn on the screen. Three high level programming languages are available for shader development:

1. NVIDIAs C for Graphics (Cg) [6]

2. OpenGLs Shading Language (GLSL) [25]

3. Microsofts High Level Shading Language (HLSL) [2]

Cg and HLSL are hardware and operating system dependent where as GLSL is an open standard and is supported on most of GPU hardware available in the market.

This project used a typical high-end commercial smartphone (Figure 2.2), a NOKIA's N900 [3] which is equipped with an OMAP3430 processor and a dedicated PowerVR SGX530 GPU supporting the OpenGL ES 2.0 API [20]. The OpenGL ES 2.0 API is a simplified version of the OpenGL API that has been designed for handheld devices. OpenGL ES 1.0 supports both fixed and programmable rendering pipeline but 2.0 only supports the programmable pipeline, meaning that developers are required to create shaders to render graphics. The N900 runs the Maemo operating system [4] which is a Linux operating system based on the GNU/Linux and GNOME/GTK+ open source technologies and developed for handheld devices.

---

[2] http://msdn.microsoft.com/en-gb/directx/

[3] http://maemo.nokia.com/n900/specifications/

[4] http://maemo.org/

Figure 2.2: The NOKIA's N900 smartphone. (*Photograph courtesy of www.fonearena.com*)

2.2 Computing on Programmable GPUs

2.2.1 General Purpose Computing on Desktop GPUs

General purpose computing on GPUs [5] have drawn significant attention from researchers because the GPU's data-parallel architecture enables high performance computing. GPUs are specifically designed for computer graphics rendering, which requires data precision for colour, vector and matrix calculations. Thus, GPUs are very useful for a range of different scientific research applications. [23] have conducted a survey of GPU based applications including physics-based simulation, signal and image processing, global illumination and geometric computing. Researchers have found that GPUs are more efficient than CPUs for domain-specialized data-parallel computing. However, GPU programming for general purpose computing is relatively difficult to implement because of its graphics oriented data structure. Researchers focus on the development of common frameworks and techniques for general purpose GPU computing.

NVIDIA [6] has developed a compiler and set of programming tools for general-purpose programming on its GPUs. The tools are known as CUDA

[5] http://gpgpu.org/

[6] http://www.nvidia.com/

(Compute Unified Device Architecture) [7]. A similar GPU focused SDK, BrookGPU [8], was developed by the Stanford University Graphics Lab. Their aim is to help software developers to implement applications on GPUs. These SDKs are designed for general purpose computing on the GPU and do not target any particular application domains.

2.2.2 Image Processing on Desktop GPUs

Farrugia [5] has developed a GPU based image processing library. This library offers an Intel OpenCV-like programming interface to a high level GPU accelerated image processing library. Another GPU based computer vision library is OpenVidia [8], based on nVidias CUDA SDK and using Cg and OpenGL for shader programming. MinGPU [3] is another light weight GPU library for computer vision, providing homography transformations, Lukas-Kanade optical flow, correlation filters, pyramids, convolutions, and Gaussian filters on a GPU.

Other researchers have worked out different ways to take advantage of graphics hardware acceleration in their computer vision applications. For example, Ready and Taylor [24] used the GPU to enhance real time feature tracking. In feature tracking, in order to achieve a more accurate tracking result, a large set of features are required. However, this increases the pressure on the CPU and reduces the performance of the tracking application. Therefore, by moving the tracking algorithm onto the GPU and carefully designing the communication data flow between the GPU and the CPU there can be a significant improvement in both the accuracy-level and application speed. Similar GPU focused enhancements in computer vision tracking applications can be also found in [18] and [28]. One common problem they tried to solve is how to balance the data communication between the GPU and the CPU because downloading data from the GPU can be relatively slow.

---

[7] http://www.nvidia.com/object/cuda_home_new/

[8] http://graphics.standford.edu/projects/brookgpu/

2.2.3 Image Processing using Mobile GPUs

There has been little research done on mobile GPU-based image processing on smartphones. One reason for this is that developer APIs for programmable mobile GPUs were not commercially available until late 2009. Smartphones with GPU hardware were already available in the market but without development kits third party developers could not access the GPUs. However, there has been some relevant research completed for example [21] has implemented Scale Invariant Feature Extraction (SIFT) on the GMA X3100 GPU [9] which is a mobile GPU for laptop computers. The study showed that the mobile GPU had performance gain for steps that can take advantage of data parallelism such as feature descriptor generation.

There are two useful mobile GPU emulators, Mali managed by ARM [10] and POWERVR SDK managed by Imagination Technology [11]. Both emulators support OpenGL ES 2.0 and can be used in different operating systems with OpenGL compatible graphics cards. However, the emulators cannot provide an accurate performance measurement because they redirect API calls to host computers. Therefore, a preliminary experiment has been carried out in this project to compare CPU and GPU-based image processing on smartphones. Chapter 3 will discuss the experimental results.

2.3 Limitations of Image Processing on GPUs

GPUs are specially designed for graphics rendering so they are fast to load data but can be slow to transfer data back to the CPU. This section addresses some issues when using GPUs for image and video processing.

Fragment shaders can process single pixels simultaneously in an image. However, two major issues need to be addressed:

1. Each fragment shader can only operate on its own pixel and no neighbourhood pixels can be accessed.

[9] http://en.wikipedia.org/wiki/Intel_GMA

[10] http://www.malideveloper.com/

[11] http://www.imgtec.com

2. Reading data from the GPUs internal memory is relatively slow.

The first issue can be solved by storing the source image as a texture in the GPU memory. Then the fragment shaders can access the shared texture memory at any time. If the fragment shader knows the position of the current pixel the position of the neighbour pixels can be computed and their value can be retrieved from the stored texture.

The second problem can also be solved by using texture memory. Instead of downloading the convoluted image from the GPU, the result image can be store in the GPU as a texture. This allows the result image to be used as an input texture for next processing kernel.

Embedded GPUs have some extra restrictions:

1. Limit the number of programming control flow statements. For example, OpenGL ES must have a known number of iterations in a loop during compilation [20] because embeded GPUs require simplified instructions to run fast on limited hardware.

2. Different implementations of OpenGL ES from manufactures have different maximum numbers of textures allowed in GPU memory.

2.4 Summary

This chapter introduced the concept of GPU-based general purpose computing. Several toolkits have been developed on specific graphics hardware. Computer vision and image processing researchers focus on GPU-based image processing and even on embedded GPUs. There are known limitations of using GPU for image processing because of the hardware architectural design. Researchers have found solutions to overcome these issues.

Chapter III

CPU vs GPU Image Processing Performance on Smartphones

This chapter presents results from a preliminary experiment comparing CPU and GPU-based image processing implementations. The experiment was carried out on a GPU-enabled smartphone.

Fragment shaders can not read the computed result of neighbourhood pixels during execution neither can they write data into the neighbourhood pixels because of the GPU's parallel computing architecture . Thus, algorithms that require global variables of a source image are relatively difficult to be implemented in fragment shaders.

However, some convolution algorithms can be easily adapted by fragment shaders:

1. Image Thresholding (binary image)

2. Edge Detection

3. Image Sharpening or Blurring

3.1 Experiment Setup

Hardware

The experimental smartphone is a NOKIA N900. Table 3.1 outlines the hardware specifications.

Software

An OpenGL/C++ software application was developed for this experiment on N900's embedded Linux OS.

Table 3.1: NOKIA N900 Smartphone Specification

CPU	ARM Cortex A8 600 MHz
GPU	PowerVR SGX530 graphics
Camera	WVGA(848 x 480)@25fps
DISPLAY	800 x 480 pixels, 3.5 inches
OS	Maemo 5 (http://maemo.nokia.com/)
Memory	32 GB storage, 256 MB RAM

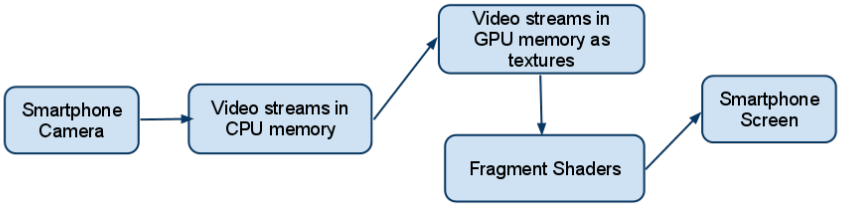


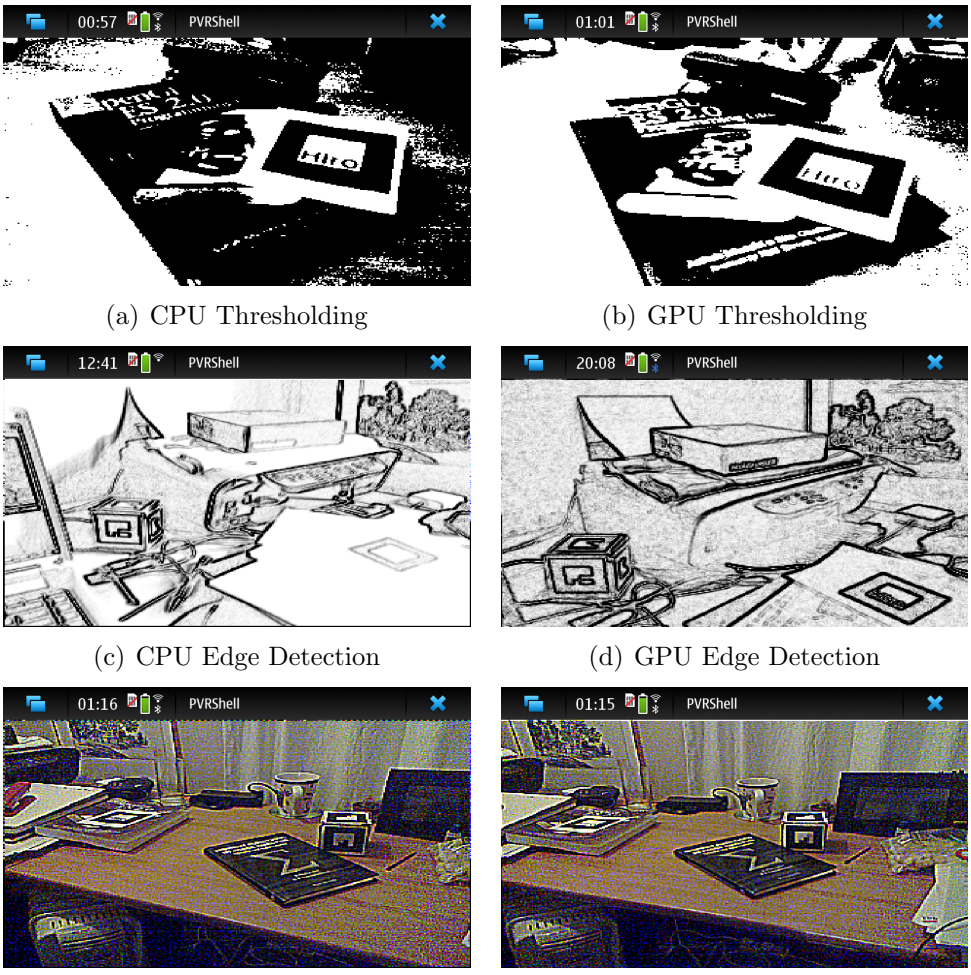
Figure 3.1: The experimental application workflow.

Method

It is difficult to measure a shader execution time on a GPU because shaders are loaded into the hardware and no direct measurement can be performed inside the GPU by application developers. Thus, we developed a simple real-time computer vision application which captures video streams and uses the OpenGL ES 2.0 API to display the images on the device screen. The images are processed by fragment shaders using different algorithms. Frame rates of CPU and GPU-based implementations are measured.

Figure 3.1 illustrates the experimental application's workflow. The captured images are uploaded into GPU's memory as 2D textures and are displayed as a quadratic texture mapping.

Three basic image processing algorithms, including image thresholding, Sobel edge detection and Mean Removal sharpening, have been implemented in GLSL fragment shaders. The source code can be viewed in Appendix A. Examples of resulting images after applying different algorithms are displayed in Figure 3.2.



(a) CPU Thresholding

(b) GPU Thresholding

(c) CPU Edge Detection

(d) GPU Edge Detection

(e) CPU Sharpening

(f) GPU Sharpening

Figure 3.2: CPU vs GPU processed image results

3.2 Comparing CPU and GPU-Based Implementations

Figure 3.3 shows the frame per second (FPS) for three different image processing algorithms implemented on the CPU and GPU. The GPU-based implementations did not show significant performance improvement except for image sharpening. The algorithms displayed in Appendix A have generalised implementations which have not been tuned to achieve the maximum performance.

Appendix A.1 has assigned the variable *threshold* to be 0.155. The value was determined by experiments under a normal indoor lighting condition. The variable *total* is computed using RGB to luminance weighted conversion model in 3.1.

$$W_R = 0.299, W_G = 0.114, W_B = 0.587 \tag{3.1}$$

Appendix A.2 applies 3x3 Sobel edge detection kernels in 3.2. The fragment shader reflects the kernels in the implementation.

$$S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{3.2}$$

Appendix A.3 uses mean removal sharpening operator in 3.3. The fragment shader reflects the operator in the implementation.

$$M = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{3.3}$$

3.3 Comparing Different Image Resolutions

Figure 3.4 shows that the GPU-based implementation of thresholding is faster than the CPU-based one but its performance degrades while the image resolution increases. The experiment tested a resolution of 800 by 480 pixels which is the largest resolution of the N900 smartphone.

Figure 3.5 shows the performance of the GPU remains constant when

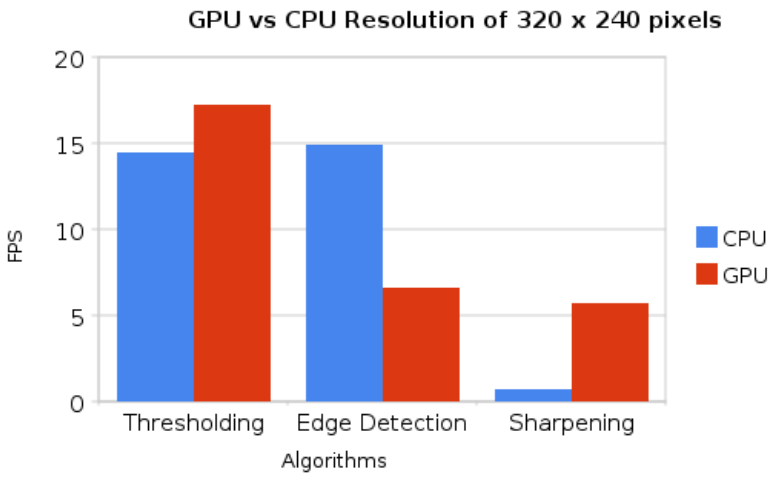


Figure 3.3: GPU vs CPU performance for 320 x 240 images

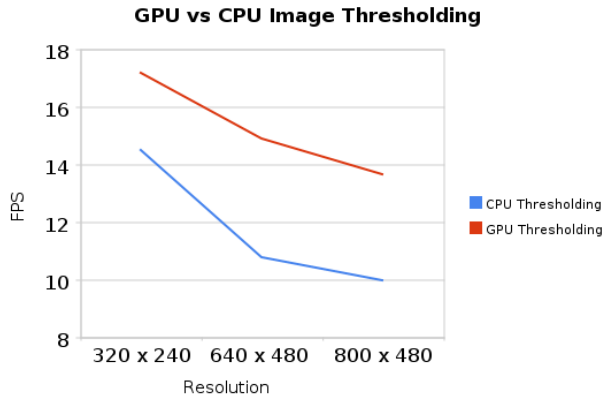


Figure 3.4: GPU vs CPU-based image thresholding implementations for a variety of resolutions

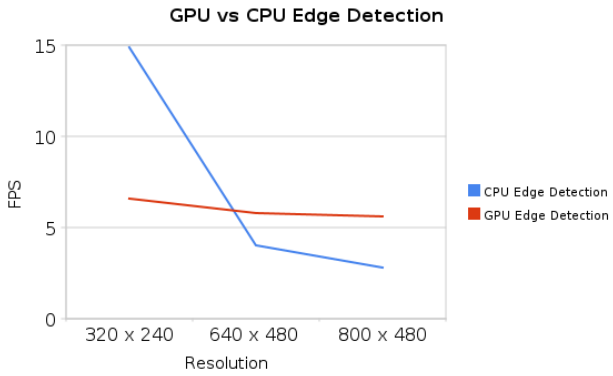


Figure 3.5: GPU vs CPU-based Sobel edge detection implementations for a variety of resolutions

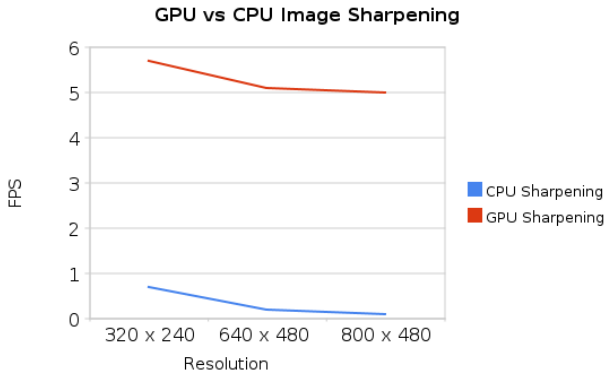


Figure 3.6: GPU vs CPU-based Mean Removal image sharpening implementations for a variety of resolutions

resolution increases. However, the CPU is more efficient for small images e.g. 320 by 240 pixels.

Figure 3.6 shows that the GPU is six times more efficient than the CPU for this sharpening algorithm and remains constant when the image resolution increases.

3.4 Comparing Combination of Algorithms

This experiment tested a combination of algorithms on the video streams because practical computer vision applications often use several algorithms to pre-process an image. The GPU-based application uses a FrameBuffer

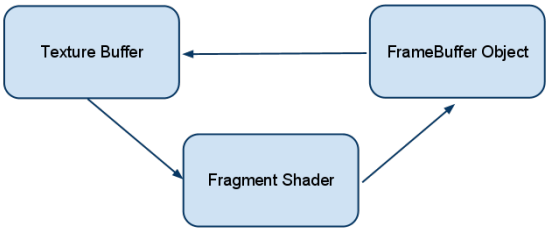
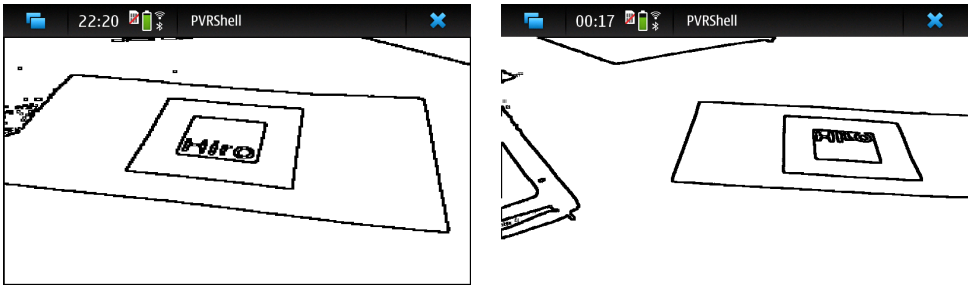


Figure 3.7: The Ping-Pong technique used by GPU-based implementations



(a) CPU (b) GPU

Figure 3.8: The final result image after thresholding and edge detection

object which allows the shader to render the resulting image into a texture buffer before displaying it on the screen. The buffered result image can be also used as an input image for the next rendering. This is the so-called Ping-Pong technique (Figure 3.7) which has been widely used in GPU computing.

This experiment ran a combination of image processing algorithms. The application first binarised a source image then applied the Sobel edge detection algorithm. Figure 3.8 shows the final results of CPU and GPU based implementation.

The tests were run over different video stream resolutions. Figure 3.9 shows the frame rate of CPU and GPU-based implementations. The CPU has a better performance than the GPU when processing relatively small image resolution. The GPU's performance remained constant over different resolutions.

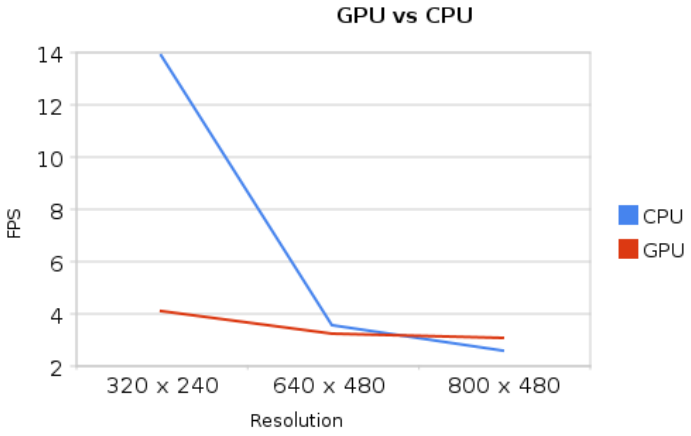


Figure 3.9: Comparison of a variety of resolutions

3.5 Summary

When applications require multiple algorithms, the CPU-based implementation is faster than the GPU-based equivalent on relatively small resolutions e.g. 320 by 240 pixels. The reason for this could be that the C++ program has been optimised by the compiler. However, the GPU performs better for complex algorithms such as image sharpening over different image resolutions.

Chapter IV

Improving Thresholding Using Image Histogram

This chapters demonstrates our new techniques to improve image thresholding for the ARToolKit library. The optimised binary images can help the ARToolKit marker detection method produces a better result than using a constant thresholding value. The new methods are based on image histogram and histogram equalisation. A GPU-based implementation for the histogram generation is also presented. Finally the new methods are integrated into the ARToolKit pipeline and experiments were carried out on the N900 smartphone.

4.1 Related Work

Image thresholding is a fundamental step in the ARToolKit marker detection method because a good binary image can improve object segmentation. The ARToolKit library uses a constant value that works well under a fixed AR application setup. For an indoor desktop AR application users can manually adjust the lighting conditions and changing video camera exposures. However, for mobile AR applications users can use smartphones under different lighting conditions and even in an outdoor environment. The users might not be able to adjust the camera settings manually through hardware or software interfaces.

ARToolKitplus [30] proposed a solution based on calculating a median of a detected marker's pixels from previous frame and set the median as the threshold value for the next frame. One of the downsides of this method is that when there is no marker detected in the image it first takes a random guess and requires a few frames to find the correct median until a marker is detected.

(a) Original Image (b) Binarized Image

Figure 4.1: Example of a poorly binarized image with thresholding value 80

4.2 Our Modified Histogram Equalization-Base Image binarisation

The ARToolKit library uses a black printed marker as a reference object to estimate the camera's position and orientation in real world. The reason for using the high contrast of black on a white background is that it is relatively distinguishable among other objects in a scene, given ARToolKit's monochrome image processing. However, different illuminations can affect the image thresholding process (see Figure 4.1).

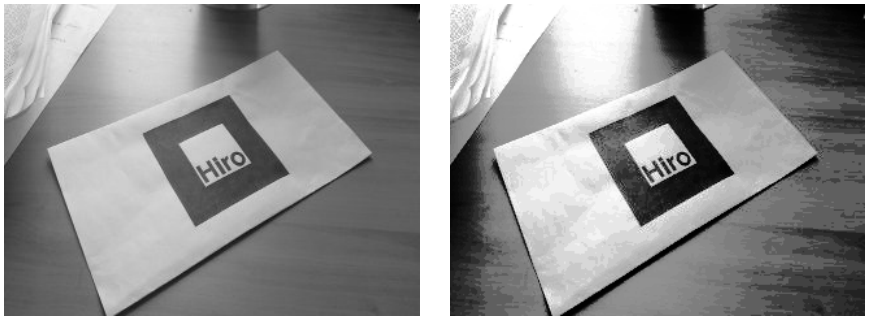
Histogram Equalization is a process to enhance image contrast based on a cumulative frequency. It re-distributes the frequency over the entire histogram bins equally. For each pixel in the image Equation 4.1 is applied.

$$E(x, y) = cumulativeFrequency[I(x, y)] \times (L \div ImageSize) \tag{4.1}$$

where *I(x, y)* is the source pixel value and *E(x, y)* is the equalised pixel value. L is the grey scale level; usually set to 255. *cumulativeFrequency* is calculated from the source image histogram.

Figure 4.2 shows a histogram equalized image that has a higher contrast than the original image and the marker has become much darker.

After thresholding the equalized image using a thresholding value of 80 Figure 4.3 shows a better segmented marker. Thus, histogram equalized images can produce a better binarized result than using the original image.



(a) Non-equalized Image (b) Equalized Image

Figure 4.2: Example of Histogram Equalization

(a) Binarized image with no equalization (b) Binarized image with equalization

Figure 4.3: Image thresholding with/out histogram equalization

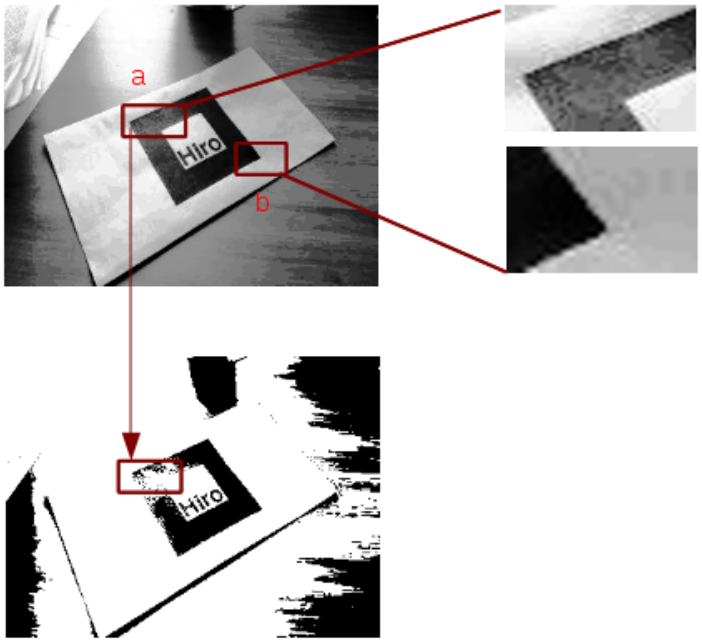


Figure 4.4: Example of falsely binarized pixels

4.2.1 Modified Histogram Equalization

Global histogram equalisation can improve the binarisation process significantly but some areas of a marker can still have higher illumination than other areas (see Figure 4.4). Area a and b are both part of the object but area a was selected as background pixels because of the high local illuminations. These background pixels in the marker can make it difficult to detect the ARToolKit marker because the ARToolKit labelling process searches connected components.

To overcome this issue the histogram equalisation equation 4.1 can be modified to include a scale factor which increases the image contrast i.e. if a pixel is below the median of grayscale level (for example 125) it will be set darker otherwise it will be set brighter. In Equation 4.2 n is the scale factor and L is the grayscale level (normally 255).

(a) No equalization (b) Unmodified equalization

(c) Modified equalization with n = 2

Figure 4.5: Binarized images

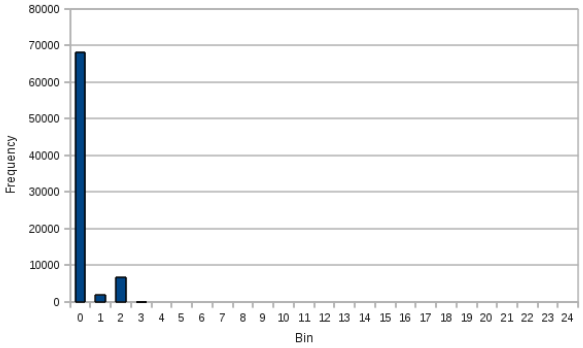
$$E(x,y) = \begin{cases} cf[I(x,y)] \times (L \div imageSize) \div n & if I(x,y) < \frac{L}{2} \\ cf[I(x,y)] \times (L \div ImageSize) \times n & if I(x,y) \geq \frac{L}{2} \end{cases} \quad (4.2)$$

In Figure 4.5 result c shows the modified equalization was applied on a image and then the image was binarised with the same thresholding value of 80. Note that although the result c now has more connected components detected (black areas) the ARToolKit labelling process will reject components with a extremely large size.

4.3 Our Image Histogram-Based Automatic Threshold

The modified Histogram Equalization can improve image binarisation with a constant thresholding but in some extreme lighting conditions the thresholding value must be adjusted in order to produce a good result. Figure 4.6 shows a marker placed in a very dark environment and its histogram.

(a) A marker in an extremely dark environment



(b) Histogram of the image a)

Figure 4.6: An example of an extremely dark lighting condition

Algorithm 1 Search for a best thresholding value

```
for each frame do
  using I ∈ [bin_min, bin_max] to threshold the frame
  if marker found then
    I is cached
  else
    increase I and go to next frame
  end if
end for
```

Our approach is to use the image histogram to automatically determine a possible range of threshold values then using a test-and-try technique to narrow down the range until a best value is found. The method uses a reduced bin size such as 25 bins rather than 255 as shown in Figure 4.6 a). Each bin represents a pixel range, bin 0 has the range from 0 to 10 and bin 24 includes all pixels greater than 250. An advantage of using reduced bin size is that an estimated range can be quickly narrowed down rather than trying all possible values from 0 to 255.

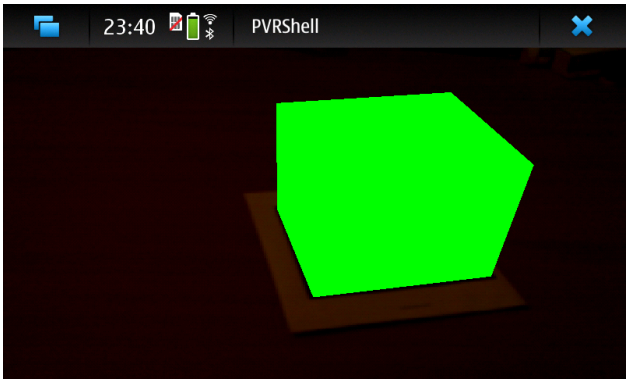
Now the best possible thresholding value can be found within the selected range starting from the lower boundary of the bin (Algorithm 1).

Figure 4.7 demonstrates two examples of using the above method to detect markers in extreme conditions. A green cube was drawn on the detected marker.

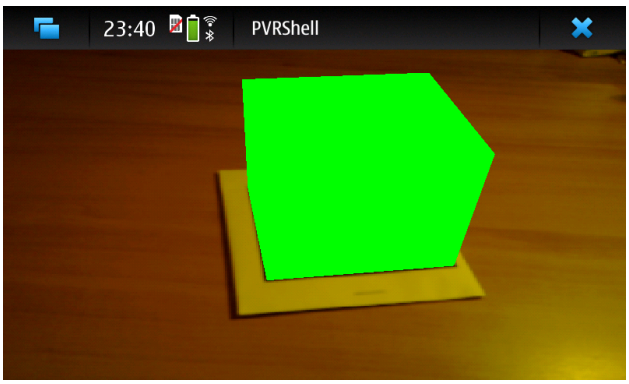
Due to the constrained computing power on somartphones histogram generation can be optimized to use a sub-image or range of interest (ROI) rather than a full sized image to reduce computational expense. Especially for single marker oriented AR applications the ROI usually can be defined around the centre of an image where the marker will be likely to be presented.

4.4 Our GPU-Based Image Histogram Generation

Histogram generation can be computationally expensive especially on resource constrained devices. The GPU as a co-processor of CPU can share the computation load. This section explains a new GPU-based method to efficiently generate the image histogram using fragment shaders and textures.



(a) Detected threshold value of 10 in a dark environment



(b) Detected threshold value of 75 in a bright environment

Figure 4.7: Examples in extreme conditions

Scheuermann and Hensley[26] have developed a desktop GPU-based image histogram generation using both vertex and fragment shaders. The method first creates vertices in a vertex shader that are the same number of pixels of a source image. Pixel values are stored as vertex attributes which are processed in by a vertex shader. For each histogram bin, pixels are moved into the same positions of the pixels having same bin values. The fragment shader takes the vertex primitives to create local histograms which can be combined to create a final histogram. One disadvantage of this method is that the vertex shader is not highly paralleled as the fragment shader.

Our new histogram generation method is similar to Scheuermann and Hensley's method that creates local histograms but only requires fragment shaders and lookup textures as bins. In the second rendering pass all local histograms are grouped together to create a global histogram. There are three key parts to this new method:

1. Texture-based local histogram bin generation

2. Texture sampling for frequency gathering

3. Global histogram generation

4.4.1 Texture-based Local Histogram Bin Generation

A procedural texture T is generated with a width of 255 and a height of an input image, 255 by 240 for example. Each row of T contains grayscale values from (0, 0, 0) to (255, 255, 255) (Figure 4.8). The reason to use this image as a lookup texture is that the fragment shader position *(x, y)* is the device's screen coordinates not a discrete position. Thus, the pixel position value *(x, y)* can not be used as a bin.

4.4.2 Texture Sampling for Frequency Gathering

An input image is also loaded as a texture. When the fragment shader processes a pixel at a fragment position *(x, y)*, it looks up T and finds a pixel P which is (125, 125, 125) for example (Figure 4.9). Then the shader searches the same value of P in the same row of I. If a matching pixel is found,

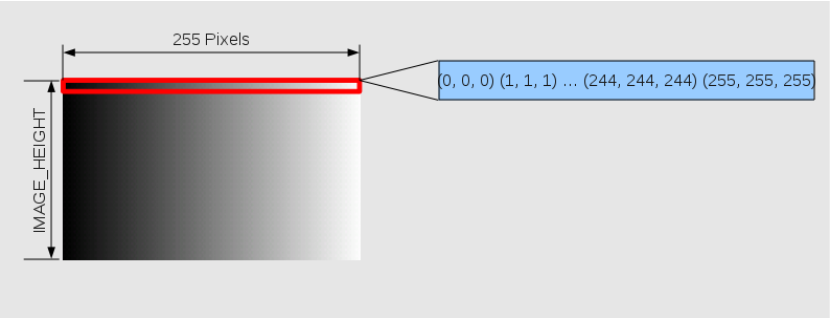


Figure 4.8: A procedural texture stores histogram bins in each row

a counter increases and is stored at fragment position *(x, y)* as *(counter, counter, counter)*. The next searched value will be (126, 126, 126).

The visualised histogram image (Figure 4.10) contains 240 local histograms created by the GPU. This result image was post-processed so that the frequency patten can be visible on the black background. The white pixels indicate the frequencies of the bins. Each column represents a bin. In this example, the source image was quite dark because high frequencies are shown between bins of 50 and 0. The magnified area shows 3 bins and 22 local histograms.

4.4.3 Global Histogram Generation

The final step is to create a global histogram using a second fragment shader to sum up all pixels of a column. The final result can be created in a 1D texture to reduce the time when transferring to CPU memory for later use in applications.

4.4.4 Mobile GPU Performance Issue

The performance of the new method will be degraded when 255 bins are used as well as when an input image width is over 255. This is because the fragment shader needs to sample 255 pixels in the source image per fragment colour calculation. The performance hit is caused by constrained memory of the embedded GPUs that the *for loop* instruction [20] has a limited number of iterations.

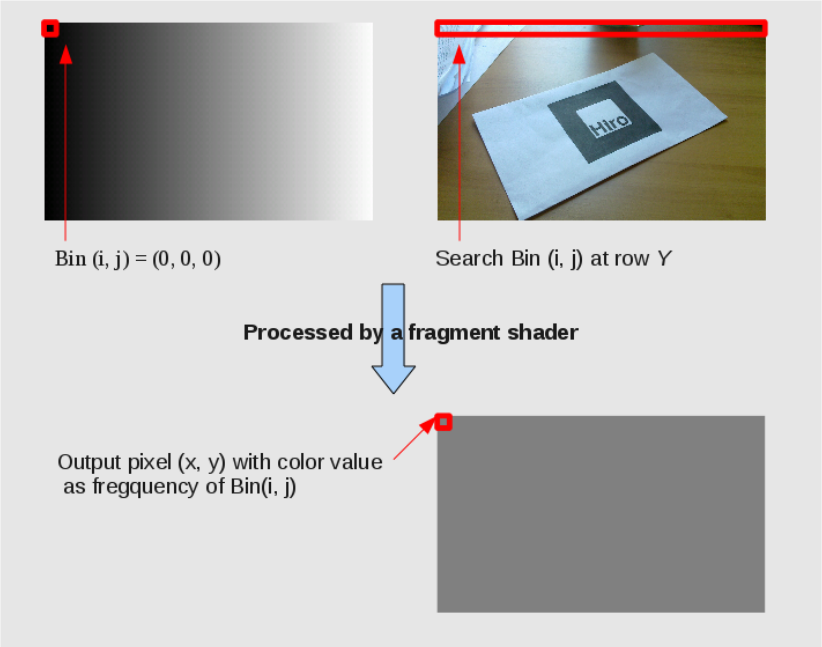


Figure 4.9: The fragment shader searches bin (i, j) at row Y of the source image and stores the counter as a pixel value at an output pixel (x, y)

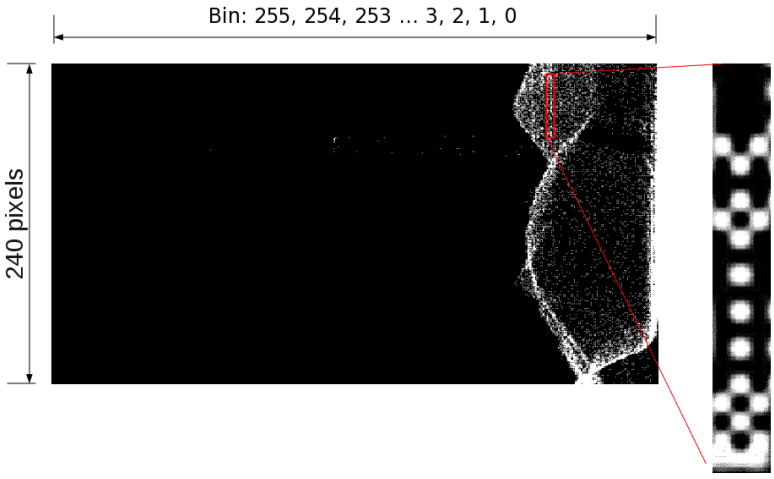
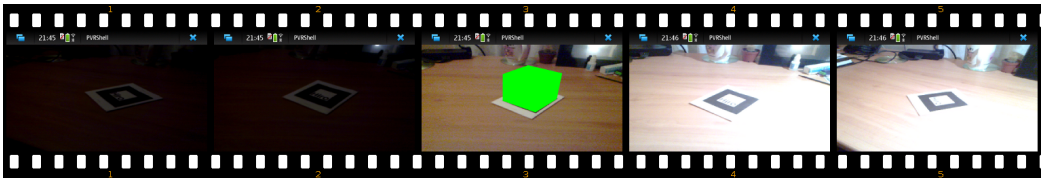


Figure 4.10: A visual example of 240 local histograms created by the GPU

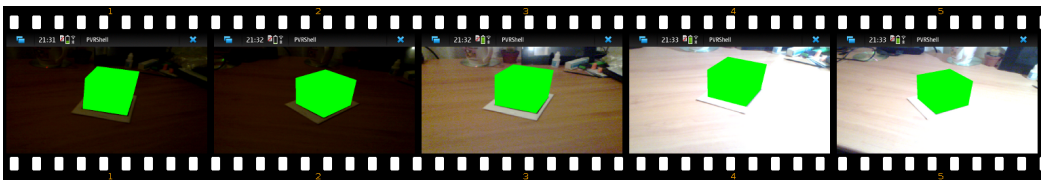
One solution is to take a sub-image from the source image. In this case its size should be smaller than 255 as smartphones have relatively small screens . Although the histogram result might not be very accurate it can still give an estimate. The sub-image should also be captured around the centre of the image because a marker is likely to be presented in front of the smartphone camera.

4.5 Comparisons of New Methods Integrated into ARToolKit

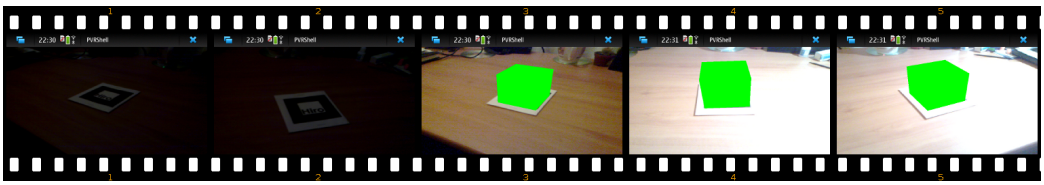
The new methods discussed above have been integrated into the ARToolKit library for comparing the marker detection results with the original AR-ToolKit library. The experiment took video streams with a resolution of 320 by 240 pixels and had a measurement matrix including the marker detection results under different lighting conditions from normal to extreme brightness as well as at different frame rates.



(a) Original ARToolKit with constant thresholding 100



(b) Histogram-based Automatic Thresholding



(c) Histogram Equalization-based Thresholding with constant thresholding 100

Figure 4.11: The video sequences illustrate the marker detection process using different thresholding methods and in normal and extreme lighting conditions.

Key frames of the video sequences for each methods are shown in Figure 4.11. The first two frames were in an extremely dark environment, the middle frame is normal condition and the last two frames are under extremely bright lighting. The Histogram-based automatic thresholding method detected 50, 120 and 240 as the threshold values for each condition.

(a) Normal condition

(b) Dark environment

Figure 4.12: The equalized results in different lighting conditions

An interesting finding was that the Histogram Equalisation-based method did not detect the marker in the dark environment. Figure 4.12 shows the equalized images during that moment.The reason for that is the dark image with a high frequency of grey-scale pixels from 0 to 20 which resulted in a cumulative frequency had high values for all bins. Thus, the Equation 4.1 produced a result close to 255 for most of the pixels.

Table 4.1 lists the performance measurement for each method. The Histogram-based automatic thresholding method has no performance loss

Table 4.1: Frame rate for the ARToolKit with different enhancement

Method	FPS
Original ARToolKit Constant Threshold	21.27
Histogram-based Auto Threshold	20.83
Histogram Equalization-based Threshold	7.57

compared with the non-enhanced ARToolKit library. The Histogram Equalization-based method has the lowest frame rate because of its image equalisation process.

4.6 Summary

This chapter presented a modified histogram equalization method which can eliminate local illuminations on the target objects. This produces good binary images which can be fed into ARToolKit labelling process. It increases the accuracy of marker detection algorithms.

In addition, an automatic threshold determination method is presented based on image histogram and overcomes the problem that when an object is under extreme lighting conditions.

Finally our novel approach to use mobile GPU for image histogram generation. This method uses fragment shaders and pre-generated textures. The performance of this implementation is better than its CPU equivalent when processing small size images with low number of bins.

The comparisons of the original ARToolKit binarization method, the histogram-based automatic thresholding and the histogram equalization based thresholding showed that histogram-based automatic thresholding produces a relatively better result under extreme and normal lighting conditions but slightly reduces the ARToolKit framerate.

Chapter V

A Fast Painterly Rendering Algorithm for Mobile AR

This chapter introduces our low complexity non-photorealistic rendering algorithm which can achieve a real-time rendering performance for mobile AR applications.

5.1 Related Work

Non-photorealistic rendering (NPR) styles have been extensively used in games, cartoon animation, and presentation graphics. Painterly rendering algorithms form a subclass of NPR algorithms using simulated brush strokes [14] [15] or sketches [10] to produce stylistic depictions of two-dimensional images. Painterly effects can also be generated using particle systems [19]. With recent advances in the areas of graphics hardware specifications and shader programming, impressive artistic effects could be generated using the GPU [29].

Interesting applications can be found on mobile devices such as painterly rendering of captured images, mobile games, puzzles and artistic rendering of 3D objects. Another emerging application area for NPR algorithms is mobile AR [12]. The increasing importance of NPR in the field of mobile graphics motivates us to develop fast and efficient methods suitable for processors with limited power and memory.

5.2 Our Method

Our method is derived from the moment-based algorithm using connected components [22]. It replaces complex data structures and procedures with simple functions that can be easily evaluated on a smartphone. For example,

recursive functions for connected component labelling have been replaced with iterative procedures involving a sequential scan. The computations of moment functions, dither images and stroke density are not performed. Instead, region boundaries are modified using a smoothing function, and edges enhanced using a weighted combination of colour values on either side of the edge. Most of the operations are performed on an index buffer containing only integers as well as using fragment shaders on mobile GPU to enhance edge colours. The method in [22] uses geometric moments as shape descriptors for mapping brush stroke images. The proposed method, however, aims to produce artistic styles using a series of simple operations on an index buffer, without generating additional brush stroke images. There are three main parts to the proposed algorithm:

1. Connected component labelling

2. Boundary extraction and smoothing

3. Region and edge enhancement

5.2.1 Sequential Connected Component Labelling

A majority of painterly rendering algorithms work on regions of nearly the same colour as an artist would paint an image by first identifying brush stroke regions of same colour. Region based painterly rendering algorithms can be found in [9], [14], [27] and [22]. Several image processing techniques, from simple to complex, can be used for region segmentation.

Due to the constrained computation power of smartphones, we try to eliminate recursive structures that take up both computational time and stack space. In the sequential method, connected components are identified by using an index buffer that stores thresholded colour indices obtained in a single top-to-bottom, left-to-right scan of the image. A colour table containing the correspondence between indices, colour values, and the total number of pixels containing each index is also maintained (5.1). The iterative algorithm [16] uses a sequential scan visiting each pixel only once, and comparing its colour value with those stored against its neighbors indices as shown in

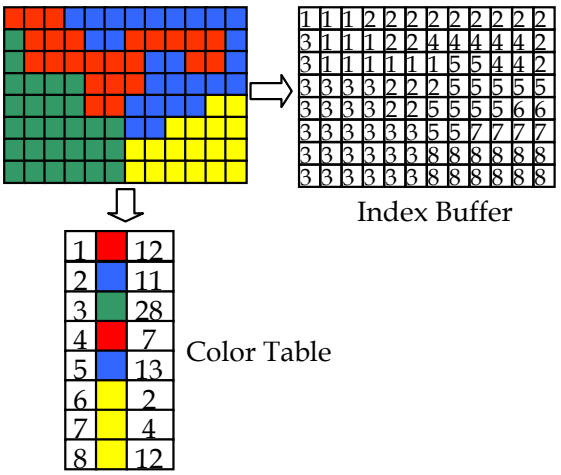


Figure 5.1: An example of colour table and index buffer produced by the sequential scan

5.2. The order in which this comparison is made affects the values stored in both the index buffer and the colour table.

It may be noted that the order of comparison also affects the way in which a single connected component is split into multiple components where either the x-monotone or y-monotone properties are not satisfied.

For all practical purposes, the sequential algorithm gives good results, except for the fact that there may exist several indices in the colour table with the same colour. This is not a major issue, as the edge generated

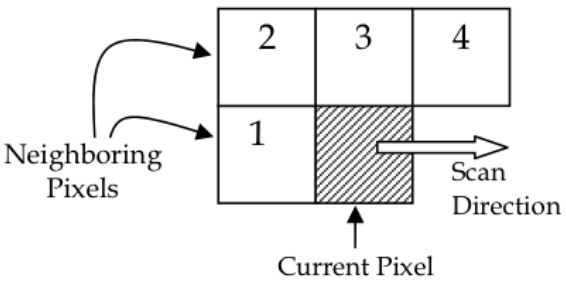


Figure 5.2: The scan order in which neighboring pixels are compared with the current pixel

between the corresponding connected components of the same colour that border each other and will not be visible. However, an undesirable artefact produced by the sequential scanning algorithm is a set of distinctly visible horizontal or vertical streaks of colours, as can be seen in 5.3.

An effective solution to this problem can be obtained by performing a *reverse scan* immediately after completing each row of the image. Moving from right to left, the index of each pixel is compared with that in the previous row, and the larger value is replaced with the smaller if the colour values match. The total number of pixels stored against the corresponding indices in the colour table is also simultaneously updated. Thus with the modified algorithm, each row is scanned twice (once in each direction), updating both the index buffer and the colour table. However, the reverse scan can be performed using only the index buffer and the colour table. The improvement in the result can be clearly seen in Figure 5.4.

The bottom-right portion of the index buffer in Figure 5.1 shows three horizontal segments corresponding to the colour yellow, that were produced by the sequential scan algorithm. The reverse scan rectifies this problem, and the modified portions of the index buffer and colour table are shown in Figure 5.4(a). The modified version of Figure 5.3(b) and (d) after performing reverse scan on the image, is shown in Figure 5.4(b) and (c).

5.2.2 Boundary Extraction

The index buffer and the colour table together form a convenient data structure for fast boundary extraction and the determination of region size. Since the index buffer contains only integers, the boundary between two regions can be easily identified using a sequential boundary following algorithm outlined below.

For each index value v, a sequential scan of the index buffer returns the first pixel position (i0 , j0) where the index is found. Since this is clearly a boundary pixel for the region of index v, we can start an iterative boundary following algorithm from this point, each time searching for the next boundary pixel. The boundary pixels are then stored in an array for smoothing, edge processing and enhancement. A flow diagram of the boundary following

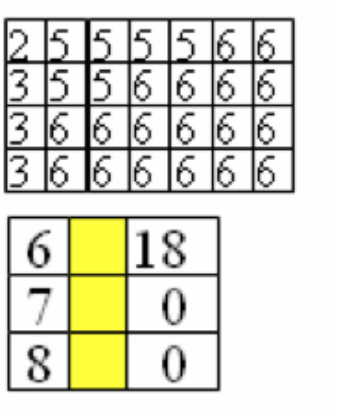
(a) Original test image

(b) Result after scan

(c) Original video frame on a smartphone

(d) Result after scan

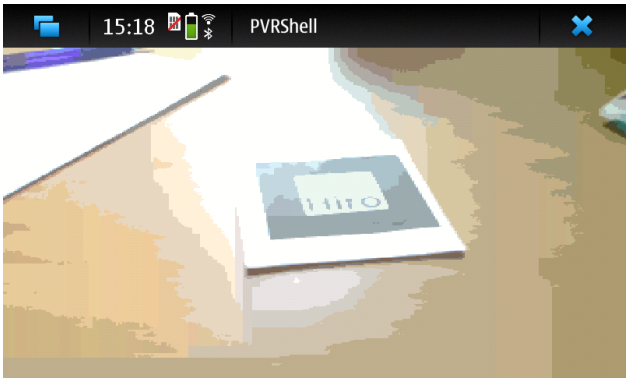
Figure 5.3: The sequential scan produces noticeable streaks. An object's colours can be blended into neighboring pixels. For instance, the white paper in image d) is blended with purple from the pen on the up-left corner.



6		18
7		0
8		0

(a) Modified colour Index Buffer

(b) Result after reverse scan



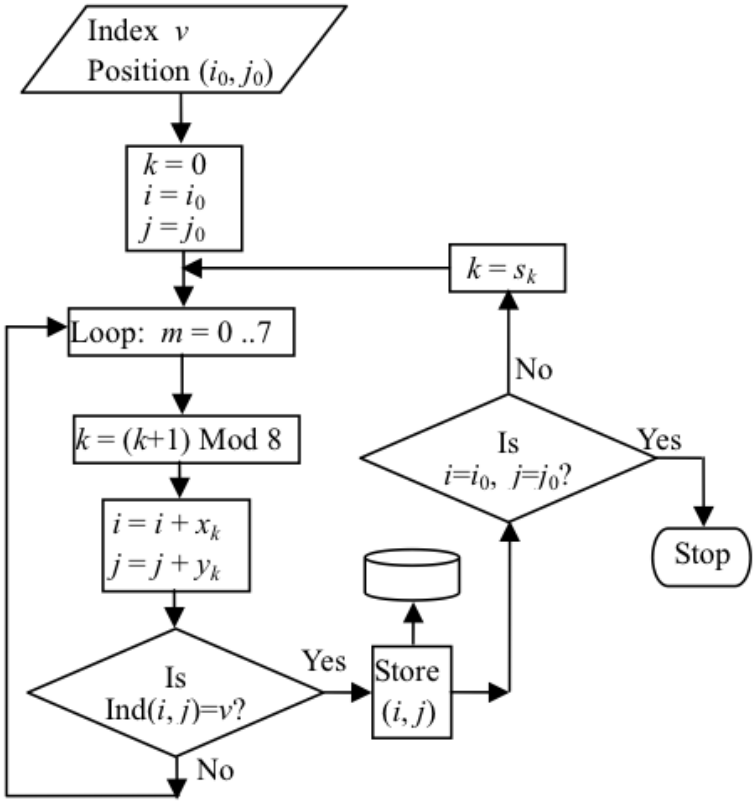
(c) Result after reverse scan

Figure 5.4: The result images after reverse scan

algorithm that traces the boundary of a region in an anti-clockwise sense, is given in Figure 5.5.

The boundary following algorithm shown above (Figure 5.5) uses an 8-connected neighbourhood for searching for the next boundary pixel, given the current pixel on the boundary. In the figure, k represents one of the eight directions, and (x_k, y_k) represents the offset vectors from the current pixel that defines the current direction (Figure 5.6). The variable s_k represents the direction in which the search starts, depending on the current direction k.

For the first identified boundary pixel (i_0, j_0), the value of k is set to 0, so that the search for the next boundary pixel in the neighbourhood of (i_0, j_0) starts from$(i_0, j_0 - 1)$ in the anti-clockwise direction. This process of



k	0	1	2	3	4	5	6	7
x_k	–1	–1	0	1	1	1	0	–1
y_k	0	–1	–1	–1	0	1	1	1
s_k	6	6	0	0	2	2	4	4

Figure 5.5: The boundary extraction algorithm

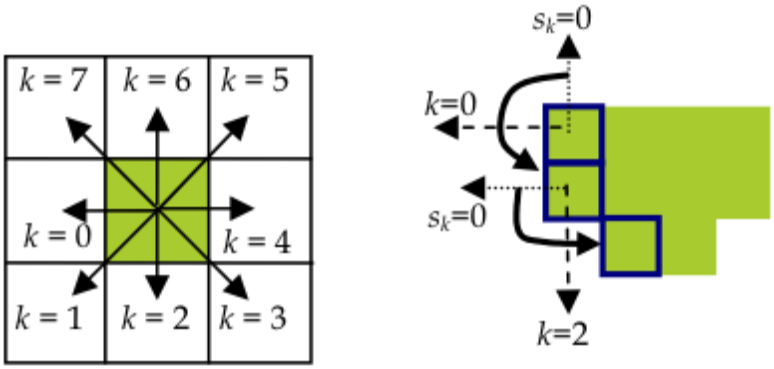


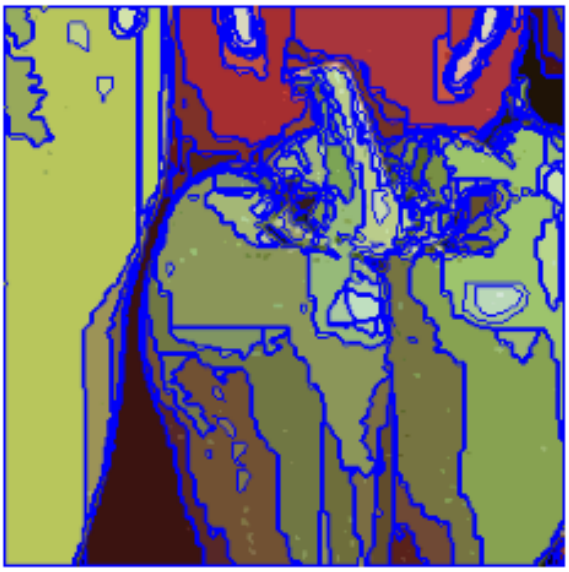
Figure 5.6: Search directions relative to the current pixel

sequentially identifying boundary pixels is illustrated in Figure 5.6. Figure 5.7 shows the test image with the boundary points marked for each connected component.

The colour values inside a region will have to be adjusted near the boundary, to match the smoothed edge. This requirement for region adjustment arises when n is large, and can also be seen clearly in Figure 5.7. Two cases to be considered are shown in Figure 5.8, with the processed region represented by green colour, and the smoothed edge in blue. Since the boundary is always traversed in the anti-clockwise sense, a downward direction in the boundary indicates a left edge, (Figure 5.8(a)) while an upward direction indicates a right edge (Figure 5.8(b)). Using this classification of edges, we easily adjust the indices of the surrounding pixels to snap the region to the smoothed edge.

The corresponding pseudo codes of the algorithm are given in algorithms 2 and 3 respectively. In both these pseudo-codes, the current region index is assumed to be v , and the current pixel position (i, j). In order to minimise the amount of redundant comparisons at this stage, it is preferable to preprocess the points on the smoothed edge, and eliminate duplicate boundary points created as a result of averaging and truncation.

In both the above cases, the total number of pixels stored against the modified indices is also simultaneously updated in the colour table. The

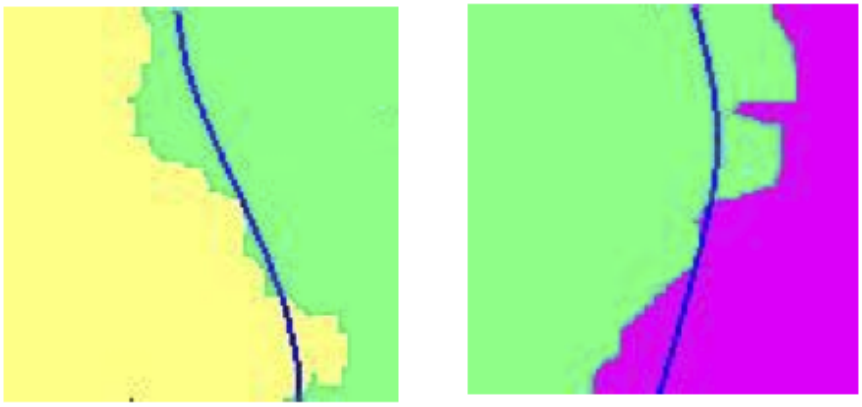


(a) Result after scan

Figure 5.7: Test image after the boundary extraction algorithm applied. The boundaries are coloured in blue.

Algorithm 2 Adjustment of region indices around left edge

```
if edge==LEFT then
  if ind(i1, j) == v then
    Scan towards left on the same row
    and get index k such that
    ind(k, j) ≠ v
    for m = k + 1...i − 1 do
      set ind(m, j) = ind(k, j)
    end for
  else if ind(x + 1, j) ≠ v then
    Scan towards right on the same row
    and get index k such that
    ind(k, j) == v
    for m = i + 1...k − 1 do
      set ind(m, j) = v
    end for
  end if
end if
```



(a) Left edge (b) Right edge

Figure 5.8: Examples of region mismatch with respect to smoothed edge

result of this operation on the original image is shown in Figure 5.9 below.

5.2.3 Region and Edge Enhancement

Depending on the colour threshold used in the connected component labelling algorithm, an image can contain several small regions or specks that need to be merged with the surrounding region. Several small patches of different colours can be seen in Figure 5.9. Region enhancement involves the process of removing these regions, by contracting the boundary to one pixel. In each iteration, the index of a boundary pixel is replaced with that of its neighbour in the index buffer, and the boundary updated. A region can be identified as *small* by checking the colour table for the total number of pixels in that region.

Unlike the previous operations, edge enhancement is done while generating the output image. For each pixel that is rendered, its index is compared with its neighbours in the index buffer. If its index is different from any of its neighbour's indices, it can be classified as an edge pixel. The colour of an edge pixel is darkened by multiplying its red, green and blue components by a value between 0.9 and 1.0.

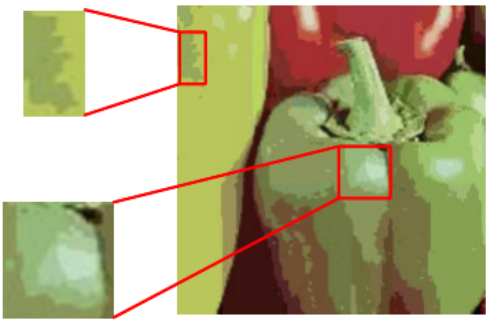
Figure 5.10 (b) shows the final image processed by all the steps described above.

Algorithm 3 Adjustment of region indices around right edge

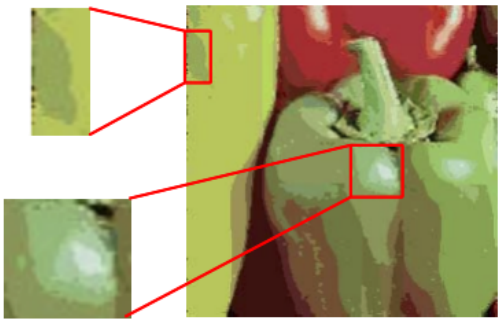
```
if edge==LEFT then
  if ind(i+1, j) == v then
    Scan towards left on the same row
    and get index k such that
    ind(k, j) ≠ v
    for m = i + 1...k − 1 do
      set ind(m, j) = ind(k, j)
    end for
  else if ind(i − 1, j) ≠ v then
    Scan towards right on the same row
    and get index k such that
    ind(k, j) == v
    for m = k + 1...i − 1 do
      set ind(m, j) = v
    end for
  end if
end if
```

5.3 GPU-Based Edge Enhancement

An alternative approach is to use fragment shaders to render edges. This method does not require any information from the previous steps such as the colour table or the index buffer so no data need to be uploaded to GPU except the image data. After the boundary extraction and edge smoothing process, the image is uploaded to GPU memory as a texture. The GPU uses the Sobel edge detection fragment shader (Appendix A2) to check if a current pixel is on an edge. For each edge pixel a new colour is assigned to improve the visual effects. Figure 5.11 shows examples of GPU-based edge enhancement in an AR application.



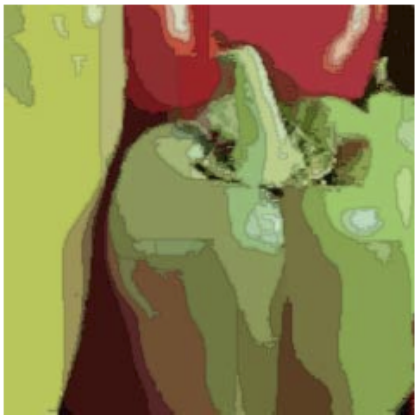
(a) Before edge smoothed



(b) After edge smoothed

Figure 5.9: An example of smoothed egdes

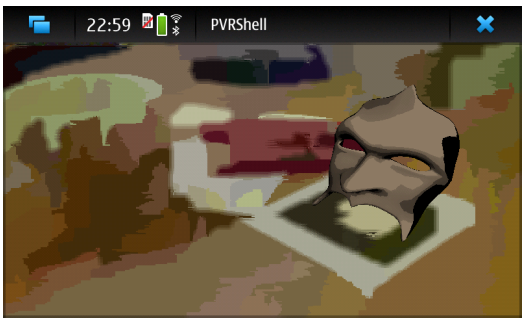
(a) Original image



(b) Processed image

Figure 5.10: The test image after all procedures applied

(a) Video frame with no edge enhancement



(b) Video frame with averaged colour on edges

(c) Video frame with dark colour on edges

Figure 5.11: Examples of edge enhancement effects. The 3D mask is rendered in Toon Shading and is presented when the ARToolKit marker is detected.

5.4 Performance Measurement

Table 5.1: FPS Measurement

Method	FPS
Original AR	32.26
Painterly Rendered AR	3.23
GPU-based Edge Enhanced AR	3.16

Table 5.1 presents the FPS of the painterly rendered AR application and the original one tested on the smartphone. The painterly rendered AR is ten times slower than the one without any effect but our new method could be further optimised by using same colour table and index buffer for consecutive frames. The FPS of the GPU-based edge enhancement and the CPU implementation are very similar. The GPU-based implementation would be recommended because the GPU has a high precision of floating point calculation, which could produce more colour variants and better visual effects.

Chapter VI

Conclusions and Future Work

6.1 Conclusions

This research project focuses on improving the user experience of mobile AR applications by introducing our methods for image binarisation and our fast non-photorealistic rendering algorithm on smartphones.

In addition, we carried out a preliminary experiment to compare CPU and GPU-based computer vision algorithms. The experimental results show that mobile GPU performed better than CPU when image resolutions are relatively large, for example 800 by 460 pixels.

The project has demonstrated our new methods to improve image binarisation based on image histogram and histogram equalization. The first method uses image histogram to determine a range for possible thresholding values. Then an automated process will pick the best value for image binarzation. This process overcomes the illumination issue for the marker detection process in the ARToolKit library. It allows mobile AR application users to move into different lighting conditions and still see 3D objects rendered on the marker. The second one uses image histogram equalisation to improve the contrast of source images. The modified version increases contrast by n times so that pixels can be further separated into extreme bins.

Our new GPU-based histogram generation method is also introduced. The method uses local histograms and stores bins in a texture. However, this method is not as efficient as a CPU-based implementation because the GPU's shader complier does not optimise shaders very well. *For Loop* statements are unrolled during compilation and both branches of *IF* statement will be executed. For complex algorithms, the mobile GPU does not perform as fast as the CPU.

The last part of this project introduced our new fast painterly rendering algorithm for mobile AR applications. The algorithm has a low level of complexity which is suitable for running on smartphones. We also used the mobile GPU to enhance the edges of the painterly rendered video frames. This provides better immersive experience for mobile AR users.

6.2 Future Work

Chapter 3 has demonstrated that the fragment shaders can be used to implement image processing algorithms. These implementations and techniques provide foundations for more complex algorithms in future research of using GPU in different aspects of AR pipeline, such as multiple marker matching and pose estimation.

In section 4.3 our assumption was that the ROI is usually around the middle of captured images. However, for non-marker based tracking, such as natural feature tracking, the tracking objects could be near the edge of the images. Future research could focus on the tracking object recovery techniques. Fourie [7] has demonstrated 100% success with blind deconvolution of binary images although the performance of the Largest Error First Harmony Search (LEFHS) algorithm needs to be optimised for mobile applications.

Our painterly rendering algorithm discussed in chapter 5 has achieved an reasonable FPS on mobile AR applications. However, to further quantify the improvement of user experience of the rendering effects extensive user studies could be carried out.

Appendix A

Fragment Shaders

A.1 Image Thresholding

```
1  uniform sampler2D sampler2d;
2  varying mediump vec2  myTexCoord;
3  const mediump float threshold = 0.155;
4  const mediump vec4 object = vec4(1.0,1.0,1.0,1);
5  const mediump vec4 bg =  vec4(0.0,0.0,0.0,1);
6
7  void main (void)
8  {
9    mediump vec4 tmp = texture2D(sampler2d, myTexCoord);
10   mediump float total = 0.299*tmp.r + 0.587*tmp.g +
   0.114*tmp.b;
11   if(total < threshold)
12   {
13     gl_FragColor = bg;
14   }
15   else
16   {
17     gl_FragColor = object;
18   }
19 }
```

A.2 Sobel Edge Detection

```
1  uniform sampler2D sampler2d;
2  varying mediump vec2  myTexCoord;
```

```
3  const mediump float step_w = 0.003125;
4  const mediump float step_h = 0.004167;
5  void main (void)
6  {
7    mediump float sum = 0.0;
8    mediump float sum_x = 0.0;
9    mediump float sum_y = 0.0;
10   mediump vec4 tmp = texture2D(sampler2d, myTexCoord.st
    + vec2(-step_w, -step_h));
11   sum_x = tmp.r * -1.0 + sum_x;
12   tmp = texture2D(sampler2d, myTexCoord.st + vec2(0.0,
   -step_h));
13   sum_x = tmp.r * 0.0 + sum_x;
14   tmp = texture2D(sampler2d, myTexCoord.st + vec2(
   step_w, -step_h));
15   sum_x = tmp.r * 1.0 + sum_x;
16   tmp = texture2D(sampler2d, myTexCoord.st + vec2(-
   step_w, 0.0));
17   sum_x = tmp.r * -2.0 + sum_x;
18   tmp = texture2D(sampler2d, myTexCoord.st + vec2(0.0,
   0.0));
19   sum_x = tmp.r * 0.0 + sum_x;
20   tmp = texture2D(sampler2d, myTexCoord.st + vec2(
   step_w, 0.0));
21   sum_x = tmp.r * 2.0 + sum_x;
22   tmp = texture2D(sampler2d, myTexCoord.st + vec2(-
   step_w, step_h));
23   sum_x = tmp.r * -1.0 + sum_x;
24   tmp = texture2D(sampler2d, myTexCoord.st + vec2(0.0,
   step_h));
25   sum_x = tmp.r * 0.0 + sum_x;
26   tmp = texture2D(sampler2d, myTexCoord.st + vec2(
   step_w, step_h));
27   sum_x = tmp.r * 1.0 + sum_x;
```

```
28    tmp = texture2D(sampler2d, myTexCoord.st + vec2(-
    step_w, -step_h));
29    sum_y = tmp.r * -1.0 + sum_y;
30    tmp = texture2D(sampler2d, myTexCoord.st + vec2(0.0,
    -step_h));
31    sum_y = tmp.r * -2.0 + sum_y;
32    tmp = texture2D(sampler2d, myTexCoord.st + vec2(
    step_w, -step_h));
33    sum_y = tmp.r * -1.0 + sum_y;
34    tmp = texture2D(sampler2d, myTexCoord.st + vec2(-
    step_w, 0.0));
35    sum_y = tmp.r * 0.0 + sum_y;
36    tmp = texture2D(sampler2d, myTexCoord.st + vec2(0.0,
    0.0));
37    sum_y = tmp.r * 0.0 + sum_y;
38    tmp = texture2D(sampler2d, myTexCoord.st + vec2(
    step_w, 0.0));
39    sum_y = tmp.r * 0.0 + sum_y;
40    tmp = texture2D(sampler2d, myTexCoord.st + vec2(-
    step_w, step_h));
41    sum_y = tmp.r * 1.0 + sum_y;
42    tmp = texture2D(sampler2d, myTexCoord.st + vec2(0.0,
    step_h));
43    sum_y = tmp.r * 2.0 + sum_y;
44    tmp = texture2D(sampler2d, myTexCoord.st + vec2(
    step_w, step_h));
45    sum_y = tmp.r * 1.0 + sum_y;
46    if(sum_x < 0.0) sum_x = -sum_x;
47    if(sum_y < 0.0) sum_y = -sum_y;
48    sum = sum_x+sum_y;
49    if(sum > 1.0) sum = 1.0;
50    if(sum < 0.0) sum = 0.0;
51    sum = 1.0 - sum;
52    gl_FragColor = vec4(sum);
```

```
53 }
```

A.3 Mean Removal Image Sharpening

```
1  uniform sampler2D sampler2d;
2  varying mediump vec2  myTexCoord;
3  /*For image width 320 pixels*/
4  const mediump float step_w = 0.003125;
5  /*For image height 240 pixels*/
6  const mediump float step_h = 0.004167;
7
8  void main (void)
9  {
10   mediump vec4 sum_x = vec4(0.0);
11   mediump vec4 tmp = texture2D(sampler2d, myTexCoord.st
    + vec2(-step_w, -step_h));
12   sum_x = tmp * -1.0 + sum_x;
13   tmp = texture2D(sampler2d, myTexCoord.st + vec2(0.0,
   -step_h));
14   sum_x = tmp * -1.0 + sum_x;
15   tmp = texture2D(sampler2d, myTexCoord.st + vec2(
   step_w, -step_h));
16   sum_x = tmp * -1.0 + sum_x;
17   tmp = texture2D(sampler2d, myTexCoord.st + vec2(-
   step_w, 0.0));
18   sum_x = tmp * -1.0 + sum_x;
19   tmp = texture2D(sampler2d, myTexCoord.st + vec2(0.0,
   0.0));
20   sum_x = tmp * 9.0 + sum_x;
21   tmp = texture2D(sampler2d, myTexCoord.st + vec2(
   step_w, 0.0));
22   sum_x = tmp * -1.0 + sum_x;
23   tmp = texture2D(sampler2d, myTexCoord.st + vec2(-
   step_w, step_h));
24   sum_x = tmp * -1.0 + sum_x;
```

```
25    tmp = texture2D(sampler2d, myTexCoord.st + vec2(0.0,
    step_h));
26    sum_x = tmp * -1.0 + sum_x;
27    tmp = texture2D(sampler2d, myTexCoord.st + vec2(
    step_w, step_h));
28    sum_x = tmp * -1.0 + sum_x;
29    gl_FragColor = vec4(sum_x);
30  }
```

Appendix B

Publications

A Fast Algorithm for Painterly Rendering on Mobile Devices, Mukundan, R., Han, C. (2008) Manchester, UK: Eurographics Conference on Theory and Practice in Computer Graphics, 9-11 Jun 2008. 67-74.

The electronic version of this paper can be downloaded at: `http://www.hitlabnz.org/publications/2008-AFastAlgorithmForPainterlyRenderingOnMobile.pdf`

"Feed The Fish": An Affect-Aware Game, Obaid, M., Han, C., Billinghurst, M. Australasian Conference on Interactive Entertainment 2008. Brisbane, Australia.

The electronic version of this paper can be downloaded at: `http://www.hitlabnz.org/publications/2008-FeedtheFish-AnAffect-AwareGame.pdf`

References

[1] M. Assad, D. J. Carmichael, D. Cutting, and A. Hudson. Ar phone: Accessible augmented reality in the intelligent environment. In *In OZCHI2003*, pages 26–28, 2003.

[2] R. T. Azuma. A survey of augmented reality. *In Presence: Teleoperators and Virtual Environments*, 6:355–385, 1997.

[3] P. Babenko and M. Shah. Mingpu: a minimum gpu library for computer vision. *Journal of Real-Time Image Processing*, 3(4):255–268, 2007.

[4] A. Dunser, J. Looser, R. Grasset, H. Seichter, and M. Billinghurst. Evaluation of tangible user interfaces for desktop ar. In *The International Symposium on Ubiquitous Virtual Reality (ISUVR). Gwangju, Korea.*, 2010.

[5] J.P. Farrugia, P. Horain, E. Guehenneux, and Y. Alusse. Gpucv: A framework for image processing acceleration with graphics processors. *IEEE International Conference on Multimedia and Expo 2006*, 2006.

[6] R. Fernando and M. J. Kilgard. *Cg, the definitive guide to programmable real time graphics*. Addison-Wesley professonnal, Boston, United States, 2004.

[7] J. Fourie, R. Green, and S. Mills. An accurate harmony search based algorithms for the blind deconvolution of binary images. In *25th International Conference of Image and VIsion Computing New Zealand*, Queenstwon, New Zealand, 8, 9 November 2010.

[8] J. Fung, S. Mann, and C. Aimone. Openvidia: Parallel gpu computer vision. In *in Proceedings of the ACM Multimedia 2005*, pages pp. 849–852, November 2005.

[9] B. Gooch, G. Coombe, and P. Shirley. Artistic vision: painterly rendering using computer vision techniques. In *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, NPAR '02, pages 83–ff, New York, NY, USA, 2002. ACM.

[10] B. Gooch and A. Gooch. *Non-Photorealistic Rendering*. A.K.Peters Ltd., 88 Worcester Street Suite 230. Wellesley, MA 02482. United States, 2001.

[11] R. Grasset, A. Duenser, and M. Billinghurst. Edutainment with a mixed reality book: A visually augmented illustrative childrens book. In *Yokohama, Japan: International Conference on Advances in Computer Entertainment Technology (ACE 2008)*, 3-5th Dec 2008.

[12] M. Haller and D. Sperl. Real-time painterly rendering for mr applications. In *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '04, pages 30–38, New York, NY, USA, 2004. ACM.

[13] A. Henrysson, M. Billinghurst, and M. Ollila. Face to face collaborative ar on mobile phones. In *Proceedings of the International Symposium on Mixed and Augmented Reality (ISMAR 2005)*, Vienna, Austria, October 5th 8th, 2005.

[14] A. Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 453–460, New York, NY, USA, 1998. ACM.

[15] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin. Image analogies. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 327–340, New York, NY, USA, 2001. ACM.

[16] R. Jain, R. Kasturi, and B.G. Schunck. *Machine Vision*. McGraw-Hill, New York, United States, 1995.

[17] H. Kato and M. Billinghurst. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In *Processdings International Workshop on Augmented Reality*, October 1999.

[18] G. Klein and D. W. Murray. Full-3d edge tracking with a particle filter. *British Machine Vision Association 2007*, 2007.

[19] B. J. Meier. Painterly rendering for animation. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 477–484, New York, NY, USA, 1996. ACM.

[20] A. Munshi, D. Ginsburg, and D. Shreiner. *OpenGL ES 2.0 Programming Guide*. Pearson Education, Inc, New Jersey, United States, 2009.

[21] M. Murphy, K. Keutzer, and H. Wang. Image feature extraction for mobile processors. In *IISWC '09: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 138–147, Washington, DC, USA, 2009. IEEE Computer Society.

[22] M. Obaid, R. Mukundan, and T. Bell. Enhancement of moment based painterly rendering using connected components. In *CGIV '06: Proceedings of the International Conference on Computer Graphics, Imaging and Visualisation*, pages 378–383, Washington, DC, USA, 2006. IEEE Computer Society.

[23] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Eurographics 2005, State of the Art Reports*, pages 21–51, 2005.

[24] J. M. Ready and C. N. Taylor. Gpu acceleration of real-time feature based algorithms. *Motion and Video Computing WMVC '07 IEEE Workshop*, 2007.

[25] R. Rost. *Open GL Shading Language*. Addison-Wesley professonnal, Boston, United States, 2004.

[26] T. Scheuermann and J. Hensley. Efficient histogram generation using scattering on gpus. In *SI3D*, pages 33–37, 2007.

[27] M. Shiraishi and Y. Yamaguchi. An algorithm for automatic painterly rendering based on local source image approximation. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 53–58, New York, NY, USA, 2000. ACM.

[28] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. Gpu-based video feature tracking and matching. *Workshop on Edge Computing Using New Commodity Architectures*, 2006.

[29] D. Vanderhaeghe, P. Barla, J. Thollot, and F. Sillion. A dynamic drawing algorithm for interactive painterly rendering. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 100, New York, NY, USA, 2006. ACM.

[30] D. Wagner and D. Schmalstieg. Artoolkitplus for pose tracking on mobile devices. In *Computer Vision Winter Workshop*, 2007.